

Queue Fundamentals, Implementation and its Applications in Round Robin Scheduling

Rollin John Fernandes

Department of Computer Science and Engineering,
RV College of Engineering
Mysore Rd, RV Vidyaniketan, Post,
Bengaluru, Karnataka, 560059 - India

RESEARCH ARTICLE

ABSTRACT: Queue is an important data structure that has wide ranging applications. In this paper, we examine various operations of queues where we focus on numerous approaches to enter as well as delete elements and also display the contents of a queue. There are two major methods to implement queues, using arrays or linked lists. The benefits of these techniques are discussed along with their algorithms for the linked list implementation of queues. The major application of queues is in the scheduling of processes. Different scheduling algorithms have been designed with varying benefits. A brief overview of a few conventional algorithms is provided following which we implement two types of round robin algorithms using queues. The simple round robin algorithm makes use of a static time quantum while the smart round robin incorporates a dynamic time quantum. These algorithms are compared for different values of burst times of processes so as to show the effectiveness of smart round robin algorithm.

KEYWORDS: Queues, Linked List, Smart Round Robin, Simple Round Robin, Scheduling, Burst Time

<https://doi.org/10.29294/IJASE.9.1.2022.2556-2566> ©2022 Mahendrapublications.com, All rights reserved

INTRODUCTION

Queues are a really important data structure that works on the FIFO (First In First Out) principle. They are linear, non-primitive systems that perform insertion and deletion operations at the same end. Here, data elements are saved in a sequential manner where each element is connected to its preceding and subsequent element. Queues are further classified into various types such as simple queue, circular queue and priority queue. These different forms have their own advantages and can be utilized in different scenarios according to the need of the programmer [1].

Arrays as well as linked lists are used for implementing queues. However, the employment of arrays often results in excessive wastage of memory. Linked list, being a dynamic data structure, does not put a restriction on the size of the queue and hence is more widely used for the same [2]. The defining of a queue followed by its usage are some operations on queues. After its use, the queue is removed from memory. These operations are regularly described as follows:

- enqueue() – To insert a component to the queue
- dequeue() – To delete a component from the queue
- display() – To display the contents of the queue

In order to make these queue operations more efficient, the following functions are also used:

- peek() – returns the element at the front of the queue
- isfull() – checks if the queue is full
- isempty() – checks if the queue is empty

Here we focus on the linked list implementation of queues. First, we define a structure **node** with a member **data** to store the content and a node pointer **next** which points to the subsequent node. The implementation of queues consists of the usage of two pointers with the names **front** and **rear**. The node inserted initially is pointed to by front and rear is employed to point to the node inserted at the end. The queue is empty if the value of front or rear is NULL. This condition is checked with

*Corresponding Author: rollinjohnfernandes4@gmail.com

Received: 10.06.2022

Accepted: 27.06.2022

Published on: 01.08.2022

Rollin John Fernandes

the isEmpty() function[1].

The function of inserting an element to a queue is referred to as enqueue and the function to delete an element is termed dequeue. The algorithms for these functions in forming a simple queue have been given below [1].

Enqueue() function:

Step 1: Create a new node named temp with the given value by setting temp->data=value.

Step 2: Check if the queue has any elements by means of the isEmpty () function.

Step 3: if (isEmpty ()==TRUE), queue has no elements, set front=temp && rear=temp.

Step 4: else set rear->next=temp, followed by rear=temp

```

12 struct node
13 {
14     int index,tat,wt,bt1,bt2;
15     struct node* next;
16 };
17
18 typedef struct node* NODE;
19
20 NODE enqueue(NODE q, int bt,int index)
21 {
22     NODE temp,cur,prev;
23     temp=(NODE)malloc(sizeof(struct node));
24     temp->bt1=bt;
25     temp->bt2=bt;
26     temp->index=index;
27     temp->next=NULL;
28
29     if(q==NULL)           //to check if queue is empty
30         return temp;
31
32     if(bt<=q->bt1)         //to check if the new entry has burst time less than the first node
33     {
34         temp->next=q;
35         return temp;
36     }
37
38     prev=NULL;
39     cur=q;
40     while(cur!=NULL&&bt>cur->bt1)
41     {
42         prev=cur;
43         cur=cur->next;
44     }
45     prev->next=temp;
46     temp->next=cur;
47     return q;

```

Figure 1 A sample code for enqueue function

```

49 NODE dequeue(NODE q,int index)
50 {
51     NODE var,prev;
52     var=q;
53
54     if(q->index==index)    //to check if the first element is to be deleted
55     {
56         q=q->next;
57         free(var);
58         return q;
59     }
60
61     if(q->next==NULL)      //if there is only one element in the queue
62     {
63         free(var);
64         return NULL;
65     }
66
67     while(var->index!=index)
68     {
69         prev=var;
70         var=var->next;
71     }
72     prev->next=var->next;
73     free(var);
74     return q;
75 }

```

Figure 2 A sample code for dequeue function

Rollin John Fernandes

dequeue() function:

Step 1: Check if there are elements within the queue by calling the is empty() function.

Step 2: if (isempty()==TRUE), queue has no elements, return from the function.

Step 3: else, define a node pointer with the name temp and initialize it, temp=front.

Step 4: set front=front->next and remove temp by using free (temp).

Display () function:

Step 1: Check if the queue has any elements by calling the is empty() function.

Step 2: if(isempty()==TRUE), queue has no elements, exit the function.

Step 3: else, define a node pointer named temp, making it point to front, temp=front.

Step 4: while(temp->next!=NULL)

Display the contents of the node temp and use temp=temp->next to maneuver to the following node.

Step 5: In the end, display temp->data and terminate.

```

78 void display(NODE q,int index)
79 {
80     NODE var;
81     var=q;
82     while(var->index!=index)
83         var=var->next;
84     printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",index,var->bt2,var->tat,var->wt);

```

Fig 3: A sample code for display function

A major assumption done while forming these algorithms is that the queue will never be full. This benefit can be attained only by using linked lists as they use dynamic memory allocation.^[2] However, the number of elements is often restricted in the linked list by specifying the quantity of elements in the beginning as SIZE and using the is full() function to test the condition if ((rear == SIZE-1 && front == 0) || (rear == front-1)) for circular queues and (rear == SIZE-1) in linear queues. Moreover, since we know the exact position of adding and removing an element, the time complexity will be $O(1)$ for both enqueue and dequeue^[3].

Queues have wide ranging applications in today's world. From being used for routers and switches in networking to maintaining the playlist in media players, the concept of queues has proved highly beneficial. They are also used in handling website traffic and hardware or real-time system interrupts. The concept of semaphores as well as that of I/O scheduling makes use of various queue operations. However, the major use of queues lies in the process scheduling of an operating system^[4]. Process scheduling can be performed in different ways and various algorithms are adopted for scheduling depending on the no. of processes waiting to be executed and execution time of every process^[5]. Common scheduling algorithms include Shortest Remaining Time First (SRTF) algorithm, First Come First Served (FCFS) algorithm, Round Robin Scheduling

algorithm and Priority based Scheduling algorithm. The algorithms are compared for various sorts of processes having different burst times (time required for a process to execute on the CPU) and their average waiting and average turnaround times are calculated^[6].

The ready queue consists of all the processes which are waiting to be executed on the CPU. Turnaround time is that interval from the instant a process has been submitted to the ready queue till its completion. The CPU scheduling algorithm doesn't affect the quantity of your time during which a procedure is being executed. Only the quantity of time that a process must wait within the ready queue is affected. Sum of such phases spent waiting within the ready queue is known as the waiting time. The average of the turnaround and waiting times of all the processes are denoted as Average Turn Around Time (ATAT) and Average Waiting Time (AWT) respectively^[7]. Here is a basic overview of some scheduling algorithms.

First Come First Serve (FCFS): It's one of the simplest algorithms of CPU scheduling where arrival times are used as a parameter for execution. The process to be executed first is the one which arrives first. The AWT and ATAT produced are usually quite high^[8].

Shortest Job First (SJF): Here, job execution is carried out according to the burst times. Basically, a process that is completed earlier has a shorter burst time. However, SJF causes

processes having longer burst times to undergo starvation [9].

Priority Scheduling: This method of job scheduling is based on the priorities assigned. The processes executed earlier are the ones having higher priorities, while jobs having similar priorities are executed on a FCFS or Round-Robin basis. This algorithm often leads to indefinite waiting of low priority processes. This scheduling algorithm generally uses priority queue in its implementation [10].

2. EXPERIMENTAL METHODS

2.1 Round Robin Scheduling

The Round Robin algorithm executes jobs for specific quantities of time called time quantum. Thus, identical time quantum slots will be allotted to all arriving processes. Processes having their burst times greater than the time quantum will be interrupted and moved towards the end of the ready queue while the following process in the queue gets executed. Thus, the starvation of processes is eliminated [11]. Nevertheless, the value of time quantum majorly determines the efficiency. A large value of the time quantum will make it behave as a FCFS algorithm whereas a small value for the time quantum will result in increased no. of context switches [12].

Based on the value of the time quantum various modifications of the round robin algorithm have developed. Here, we will initially target the traditional round robin algorithm and then implement a smart round robin algorithm using queue data structure. This will allow us to compare the efficiency of each of the algorithms with regard to their turnaround and waiting times. Both these algorithms are coded with the help of linked list implementation of queues.

Before diving into the algorithms, it is important to note the assumptions made. The most important one is that all processes are assumed to arrive at the same time. The priority of processes is additionally not being considered here. Moreover, various operating system issues like context switching overhead are ignored during the calculation of various statistics. It is also necessary to understand the burst time of each process beforehand, which may not always be feasible [11].

2.1.1 Simple Round Robin

The first algorithm is the simple round robin where a variable quant is entered by the user.

Processes that have their remaining burst time less than quant's value get executed. But if that condition is not satisfied, the process gets executed for a period of quant, following which; execution is given to the ensuing process within the queue [13]. The algorithm uses enqueue operation of queues to store the information of processes and it uses dequeue operation to remove a process from the ready queue when its execution is complete.

Step 1: Enter the no. of processes (NOP) along with their burst times (bt).

Step 2: Store the burst times in a ready queue 'q' by using the enqueue function, enqueue(q, bt, index).

Step 3: Enter the value of time quantum quant.

Step 4: if (NOP>0), go to step 5.

else go to step 7.

Step 5: Check if remaining burst time (RBT) <= quant. if yes, the process gets executed completely. Otherwise, it is executed for a time period quant.

Step 6: if (RBT==0), display the details of that process, display(q, index) and remove the process from the queue, dequeue (q, index).

Step 7: Calculate AWT and ATAT of all the processes.

2.1.2 Smart Round Robin

The second algorithm is known as smart round robin algorithm. The major difference between the two algorithms lies in the way the time quantum is calculated. Smart round robin uses a dynamic time quantum that improves its efficiency. Here, processes are entered into the ready queue with the assistance of a modified enqueue function that stores values according to the increasing order of their burst time. The value of quant is estimated for every cycle of traversal of the list of processes and is equal to the average of the differences between the adjacent burst times. A variable delta is equated to half the value of quant. The Remaining Burst Time (RBT) of a process is the time it further needs in order to complete its execution and this value keeps reducing after each cycle. Processes that can complete their execution within (quant + delta) time, that is, the processes that have their RBT within (quant + delta) time are allowed to attain completion. This permits tasks with very little remaining time to complete their execution instead of waiting for an entire cycle to acquire necessary CPU time [14].

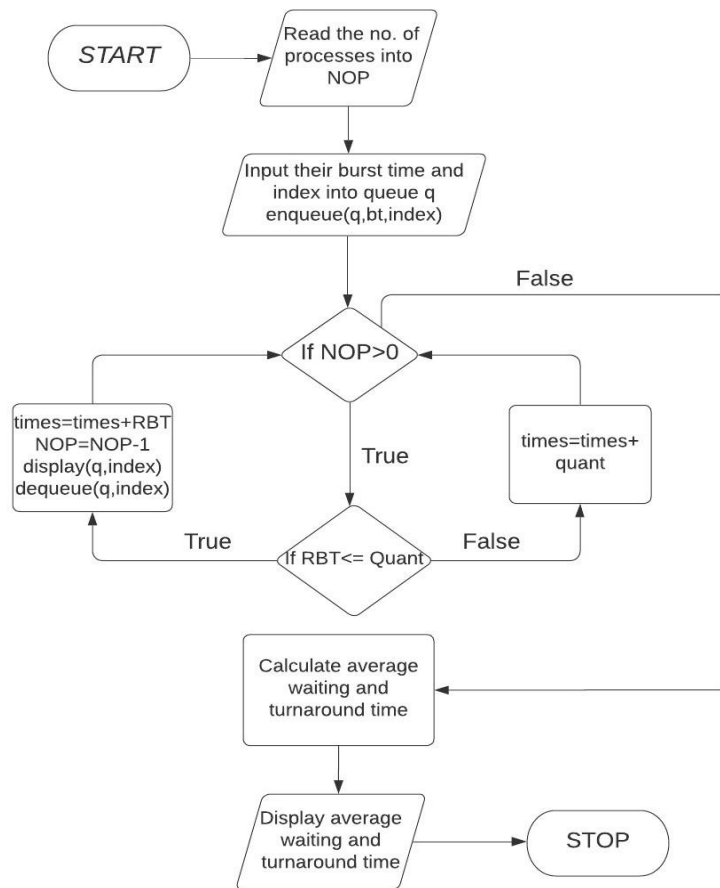


Figure 4 Flowchart for simple round robin algorithm

```

78 int main()
79 {
80     NODE q=NULL,temp;
81     int NOP,i,j,bt,sum=0,quant;
82     float avg_wt=0, avg_tat=0;
83
84     printf(" Total number of process in the system: ");
85     scanf("%d", &NOP);
86     int y = NOP; // Assign the number of processes to variable y
87
88     // Use for Loop to enter the Burst Time
89     for(i=0; i<NOP; i++)
90     {
91         printf("\nEnter the Burst time of the Process[%d]:\n", i+1); // Accept the Burst time
92         scanf("%d", &bt);
93         q=enqueue(q, bt,i+1);
94     }
95
96     printf("Enter the value of time quantum:\n");
97     scanf("%d",&quant);
98
99     temp=q;
100     printf("\nProcess No \tBurst Time \tTAT \tWaiting Time \n");
101
102     while(y>0)
103     {
104         if(temp->bt1<=quant)
105         {
106             sum=sum+temp->bt1;
107             temp->wt=sum-temp->bt2;
108             temp->tat=sum;
109
110             avg_tat=avg_tat+temp->tat;
111             avg_wt=avg_wt+temp->wt;
112             display(q,temp->index);

```

Figure 5 Main functions for simple round robin algorithm

Step 1: Enter the no. of processes (NOP) and the burst time (bt) of each process.

Step 2: Store the burst times in a ready queue 'q' such that processes' burst times are in an increasing order, by using the modified enqueue function, enqueue(q, bt, index).

Step 3: if(NOP>0), calculate the value of quant as average of difference between adjacent remaining bt. $\text{delta} = \text{quant}/2$.

else go to step 6

Step 4: Check if remaining burst time (RBT) $\leq$ quant + delta. if yes, the process gets executed completely.

else, process is executed for a time period of quant.

Step 5: if (RBT==0), display the details of that process, display(q, index) and remove the process from the queue, dequeue(q, index). Go to step 3.

Step 6: Calculate AWT and ATAT of all the processes.

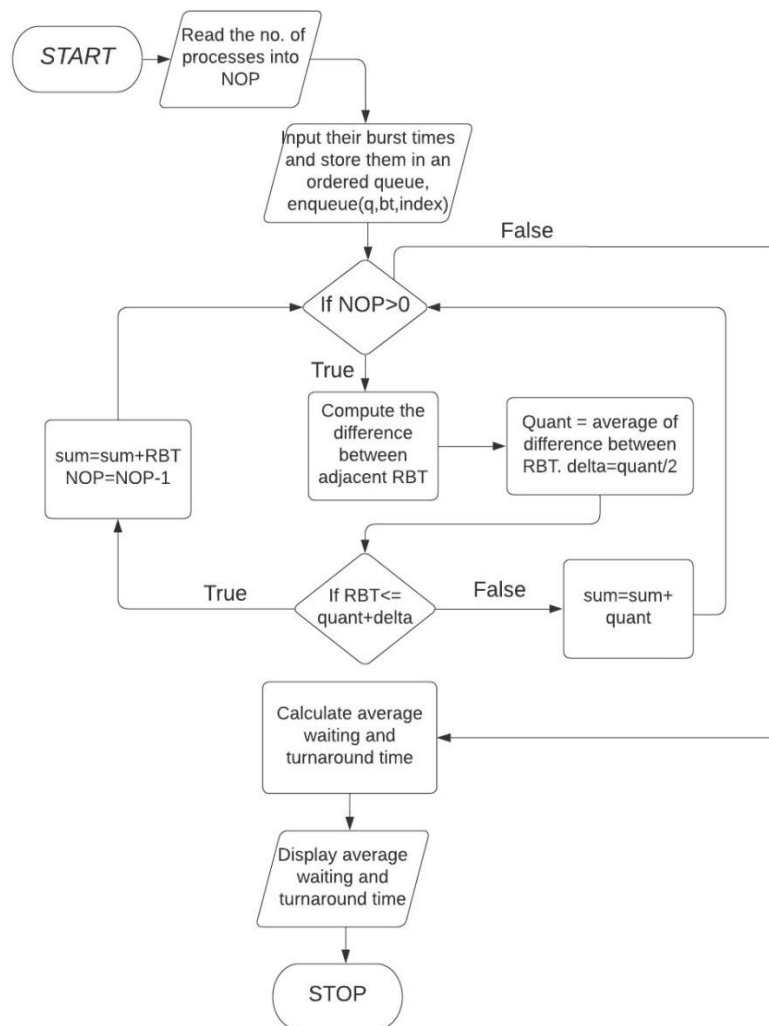


Figure 6 Flowchart for smart round robin algorithm

3. RESULTS AND DISCUSSION

The code for the above algorithms has been executed on the GDB platform using C language to obtain the following data. All timings taken within the test cases are in milliseconds (ms).

Case 1: Small burst times

In case 1, we consider small burst instances for all of the four processes. A time quantum of

2 ms is taken into consideration for the simple round robin algorithm. The outcomes are shown in the table below.

It can be observed that the average waiting and turnaround time in the smart round robin algorithm is lesser than the results obtained from the conventional methods.

```

122     if((smart->bt1-q->bt1)%(y-1)==0)
123         quant=(smart->bt1-q->bt1)/(y-1);
124     else
125         quant=(smart->bt1-q->bt1)/(y-1)+1;
126         delta=quant/2;
127     }
128
129     while(temp!=NULL)
130     {
131         if(temp->bt1<=quant+delta)
132         {
133             sum=sum+temp->bt1;
134             temp->wt=sum-temp->bt2;
135             temp->tat=sum;
136
137             avg_tat=avg_tat+temp->tat;
138             avg_wt=avg_wt+temp->wt;
139             display(q,temp->index);
140
141             NODE extra=temp;
142             temp=temp->next;
143             q=dequeue(q,extra->index);
144             y--;
145         }
146         else
147         {
148             sum=sum+quant;
149             temp->bt1=temp->bt1-quant;
150             temp=temp->next;
151         }
152     }
153
154     printf("\n Average Turn Around Time: %f", avg_tat/NOP);
155     printf("\n Average Waiting Time: %f", avg_wt/NOP);
156     return 0;
157 }

```

Figure 7 Main functions for smart round robin algorithm

Table I: Case 1: Small Burst Times

Process	Burst Time	Simple RR		Smart RR	
		WT	TAT	WT	TAT
P1	6	8	14	14	8
P2	3	7	10	1	4
P3	1	4	5	0	1
P4	4	8	12	10	6

Algorithm	AWT	ATAT
Simple RR	6.75	10.25
Smart RR	3.75	7.25

Table II: Gantt chart for Simple Round Robin

P1 0 2	P2 4	P3 5	P4 7	P1 9	P2 10	P4 12	P1 14
-----------	---------	---------	---------	---------	----------	----------	----------

Initial Sorted Order in terms of Burst Time: P3, P2, P4, P1

Initial Sorted Order in terms of Burst Time for Smart Round Robin: P3, P2, P4, P1

Table III: Gantt Chart for Smart Round Robin

P3 0 1 quant=2 delta=1	P2 4	P4 6	P1 8	P4 10 quant=2 delta=1	P1 12	P1 14 quant=2 delta=1
---------------------------------	---------	---------	---------	--------------------------------	----------	--------------------------------

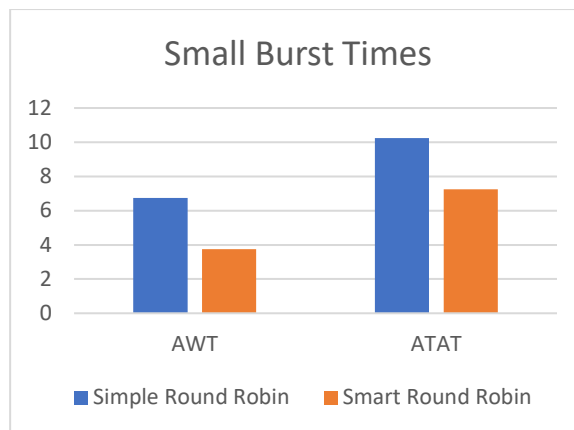


Fig 8: Graphical Comparison for Small Burst Times

Case 2: Large burst times

In the second case, we consider the burst times to be large. We assume a time quantum of **45 ms** for the jobs and acquire the statistics of waiting and turnaround times.

Table IV: Case 2: Large Burst Times

Process	Burst Time	Simple RR		Smart RR	
		WT	TAT	WT	TAT
P1	35	0	35	74	109
P2	74	151	225	178	252
P3	26	80	106	64	90
P4	89	180	269	186	275
P5	51	224	275	125	176

Algorithm	AWT	ATAT
Simple RR	127.00	182.00
Smart RR	125.40	180.40

Table V: Gantt chart for Simple Round Robin

P1 0 35	P2 80	P3 106	P4 151	P5 196	P2 225	P4 269	P5 275
------------	----------	-----------	-----------	-----------	-----------	-----------	-----------

Initial Sorted Order in terms of Burst Time for Smart Round Robin: P3, P1, P5, P2, P4

Table VI: Gantt chart for Simple Round Robin

P3 0 16 quant=16 delta=8	P1 32	P5 48	P2 64	P4 80	P3 90 quant=16 delta=8	P1 109	P5 125	P2 141
P4 141 157	P5 176 quant=19 delta=9	P2 195	P4 214	P2 229 quant=15 delta=7	P4 244	P2 252 quant=15 delta=7	P4 267	P4 275

The results obtained show the efficiency of the smart round algorithm in comparison to the simple algorithm.

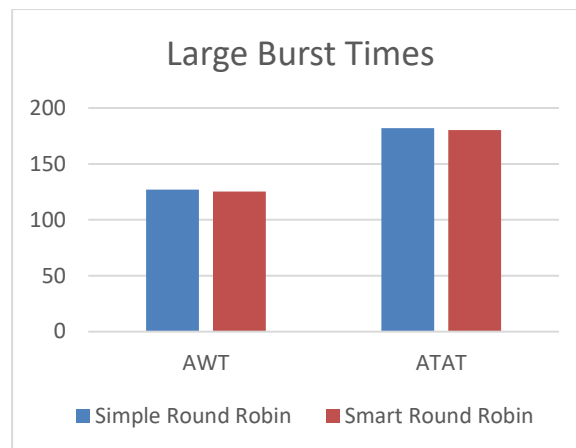


Fig 9: Graphical Comparison for Large Burst Times

Case 3: Random values

Here, we consider a random set of numbers that vary over a wide range. A time quantum of **10 ms** is assumed for the traditional algorithm.

Table VII: Case 3: Random Burst Times

Process	Burst Time	Simple RR		Smart RR	
		WT	TAT	WT	TAT
P1	12	25	37	0	5
P2	34	47	81	38	72
P3	5	20	25	5	17
P4	67	51	118	51	118

Algorithm	AWT	ATAT
Simple RR	35.75	63.25
Smart RR	23.50	53.00

Table VIII: Gantt chart for Simple Round Robin

P1 0 10	P2 20	P3 25	P4 35	P1 37	P2 47	P4 57
P2 57 67	P4 77	P2 81	P4 91	P4 101	P4 111	P4 118

Initial Sorted Order in terms of Burst Time for Smart Round Robin: P3, P1, P2, P4

Table IX: Gantt chart for Smart Round Robin

P3 0 5 quant=21 delta=10	P1 17	P2 38	P4 59	P2 72 quant=33 delta=16	P4 118
-----------------------------------	----------	----------	----------	----------------------------------	-----------

Initial Sorted Order in terms of Burst Time: P3, P1, P2, P4

The output here too shows the benefits of the smart algorithm. There is a drastic reduction in the average values attained following the execution of the two processes.

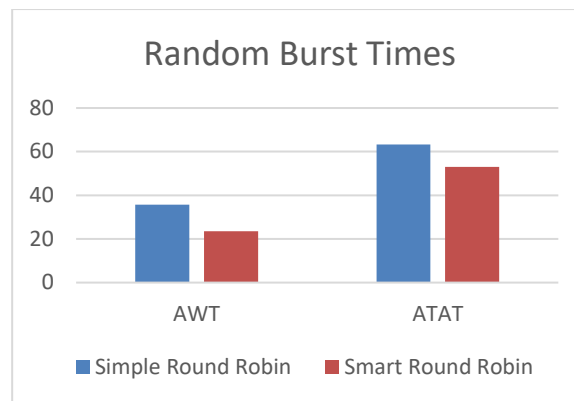


Fig 10: Graphical Comparison for Random Burst Times

3.1 ANALYSIS

The data obtained from above clearly shows the efficiency of smart round robin algorithm. Different burst times are used to test the algorithms. In every case, the reduction in the turnaround and waiting time is visible. This time saving is highly beneficial as it prevents wastage of the CPU time.

4. CONCLUSION

This study compares two primary variants of round robin scheduling and discusses several scheduling procedures. The algorithms were programmed with linked list implementation of queues. Both the algorithms were compared for different burst times of different processes and their statistics obtained led to the conclusion that the smart round robin algorithm yields better results than its traditional counterpart. In fact, smart round robin has also proven to be superior to some other conventional methods such as SJF or FCFS, as it more often than not prevents starvation.

As these algorithms use queues, they are easier to construct because they follow a straightforward FIFO technique. Unlike the array approach, queues are built using a linked list, which does not limit the number of processes. Furthermore, the design of queues with the help of linked lists saves space and reduces memory waste. We have also utilized two alternative enqueue operations, one that stores the data on a first come basis and the other that stores it in an ascending order depending on a particular field, thus forming an ordered queue. The dequeue operation too is modified to delete the desired process rather than the first job in the ready queue, which is usually the approach in the traditional dequeue functions.

There is still scope for further enhancement by considering the arrival times of different processes and even allotting those priorities. Moreover, the memory utilization by linked lists is more and they take more time for traversal. Thus, an increased number of context switches might be a major issue. To add to this, it is not possible to directly access a process using the linked list implementation of queues. Hence, the future scope is vast but currently, considering the analysis done for various test cases, the smart round robin algorithm with linked list queue implementation is a better approach for process scheduling than the simple round robin algorithm.

REFERENCES

- [1] Salaria, R.S. 2018. Data Structures & Algorithms using C, third edition, Khanna Publishing, India, pp. 275-298.
- [2] Aaron M. Tenenbaum, Yedidyah Langsam, Moshe Augenstein, 1990. Data Structures Using C, first edition, Prentice Hall, pp. 168-188.
- [3] ZhenghuaXue, Jianhui Li, Geng Shen, Yuanchun Zhou, Yang Zhang, 2011. A customized dynamic array for job scheduling queue of a large-scale server cluster, EMEIT, 2, 842-843
- [4] Silberschatz Abraham, Peter B Galvin, Greg Gagne, 2018. Operating Systems Concepts, tenth edition, Wiley India Pvt Ltd, pp. 199-246
- [5] Fiad Alaa, Mekkakia Maaza Zoulikha, Bendoukha Hayat, 2020. Improved Round Robin Scheduling Algorithm with Varying Time Quantum, EDiS. 1, 33-34
- [6] Samih M. Mostafa, Hirofumi Amano, 2020. Dynamic Round Robin CPU Scheduling Algorithm Based on K-Means Clustering Technique, Applied Sciences, 10(15), 1-2

- [7] Tagreed Balharith, Fahd Alhaidari, 2019. Round Robin Scheduling Algorithm in CPU and Cloud Computing: A review, 1, 1
- [8] Alka Pant, 2011. A Comparison between FCFS and Mixed Scheduling, IJCST, 2(2), 445-446
- [9] Abdulaziz A. Alsulami, Qasem Abu Al-Haija, Mohammed I. Thanoon, Qian Mao, 2019. Performance Evaluation of Dynamic Round Robin Algorithms for CPU Scheduling, Southeast Con, 1, 1
- [10] Rahul Mishra, Gourav Mitawa, 2021. Improved Round Robin Algorithm for effective Scheduling Process for CPU, ICICV.1, 591
- [11] Samkit Mody, Sulalah Mirkar, 2019. Smart Round Robin CPU Scheduling Algorithm for Operating Systems, ICEECCOT, 1, 310-314
- [12] Ahmed Alsheikhy, Reda Ammar; Raafat Elfouly, 2015. An improved dynamic Round Robin scheduling algorithm based on a variant quantum time, ICENCO, 1, 98
- [13] Ayush Mangukia, Mohammed Ibrahim, Soorya Golamudi, Nishant Kumar, M Anand Kumar, 2021. Improved Variable Round Robin Scheduling Algorithm, ICCCNT, 1, 2
- [14] Montek Singh, Rohit Agarwal, 2018. Modified Round Robin algorithm (MRR), ICPCSI, 1, 2832-2834